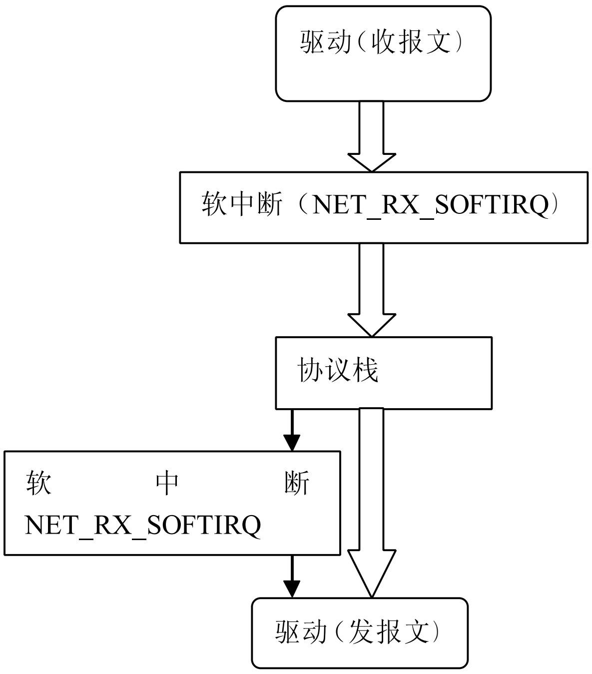
Linux 报文处理流程。

newmaker@163.com

1 目的

了解 linux 报文转发流程好比理解 Linux 网络的骨架。有个宏观的概念，对于网络开发，提高性能，bug 解决起到至关重要的作用。

2 处理一级流程图



3 软中断处理与驱动

3.1 软中断

软中断中取报文，是通过 NAPI 结构的 poll 函数取的，每个 CPU 拥有一个 NAPI 列表。、

NAPI 是驱动注册的取报文的结构。 使用 NAPI 主要是采用 poll 方式取报文，而不是依

赖硬中断处理，对于小包，报文数比较多的情况下，大大的提高的效率，毕竟硬中断太影响性能。

```
struct napi_struct {
/* The poll_list must only be managed by the entity which
  * changes the state of the NAPI_STATE_SCHED bit.   This means
  * whoever atomically sets that bit can add this napi_struct
  * to the per-cpu poll_list, and whoever clears that bit
  * can remove from the list right before clearing the bit.
  */
struct list_head       poll_list;

unsigned long          state;
int                    weight;   //为 poll 函数的第 2 个参数，表示每次期望取多少报文。如果取的报文不足该数，则从 cpu napi 队列去掉该 napi。
int                    (*poll)(struct napi_struct *, int); //处理函数。
#ifdef CONFIG_NETPOLL
spinlock_t             poll_lock;
int                    poll_owner;
#endif

unsigned int           gro_count;

struct net_device      *dev;
struct list_head       dev_list;
struct sk_buff         *gro_list;
struct sk_buff         *skb;
};
```

```
static void net_rx_action(struct softirq_action *h)
{
        struct list_head *list = &__get_cpu_var(softnet_data).poll_list;
        unsigned long time_limit = jiffies + 2;
        int budget = netdev_budget;
        void *have;

        local_irq_disable();

        while (!list_empty(list)) {
                struct napi_struct *n;
                int work, weight;

                /* If softirq window is exhuasted then punt.
                  * Allow this to run for 2 jiffies since which will allow
                  * an average latency of 1.5/HZ.
```

```
         */
/*每次软中断取报文不要高于 300，所耗时间不要超过 2 个时钟中断*/
        if (unlikely(budget <= 0 || time_after(jiffies, time_limit)))
                goto softnet_break;

        local_irq_enable();

        /* Even though interrupts have been re-enabled, this
         * access is safe because interrupts can only add new
         * entries to the tail of this list, and only ->poll()
         * calls can remove this head entry from the list.
         */
        n = list_first_entry(list, struct napi_struct, poll_list);

        have = netpoll_poll_lock(n);

        weight = n->weight;

        /* This NAPI_STATE_SCHED test is for avoiding a race
         * with netpoll's poll_napi().  Only the entity which
         * obtains the lock and sees NAPI_STATE_SCHED set will
         * actually make the ->poll() call.  Therefore we avoid
         * accidently calling ->poll() when NAPI is not scheduled.
         */
        work = 0;
        if (test_bit(NAPI_STATE_SCHED, &n->state)) {
                work = n->poll(n, weight);
                trace_napi_poll(n);
        }

        WARN_ON_ONCE(work > weight);

        budget -= work;

        local_irq_disable();

        /* Drivers must not modify the NAPI state if they
         * consume the entire weight.  In such cases this code
         * still "owns" the NAPI instance and therefore can
         * move the instance around on the list at-will.
         */
        if (unlikely(work == weight)) {
                if (unlikely(napi_disable_pending(n))) {
                        local_irq_enable();
```

```
                napi_complete(n);
                local_irq_disable();
            } else
                list_move_tail(&n->poll_list, list);    /*为了 napi 公平，移到队尾*/
        }

        netpoll_poll_unlock(have);
    }
out:
    local_irq_enable();

#ifdef CONFIG_NET_DMA
    /*
     * There may not be any more sk_buffs coming right now, so push
     * any pending DMA copies to hardware
     */
    dma_issue_pending_all();
#endif

    return;

softnet_break:
    __get_cpu_var(netdev_rx_stat).time_squeeze++;
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);
    goto out;
}
```

3.2 驱动向 CPU 注册 NAPI

以 e1000e 为例子，e1000e 在硬中断处理函数中注册 NAPI. E1000e 芯片可以动态调整硬中断的频率根据收报文率。单位时间内接收的较多的报文时，芯片要降低中断发生的次数，通过 NAPI poll 去处理取报文。

3.2.1 注册 NAPI：

```
static irqreturn_t e1000_intr(int irq, void *data)
{
      ………………
if (napi_schedule_prep(&adapter->napi)) {
        adapter->total_tx_bytes = 0;
        adapter->total_tx_packets = 0;
        adapter->total_rx_bytes = 0;
```

```
        adapter->total_rx_packets = 0;
        __napi_schedule(&adapter->napi);
    }

}
```

3.2.2 E1000e 的 POLL 函数

```
static int e1000_clean(struct napi_struct *napi, int budget)
{
    struct e1000_adapter *adapter = container_of(napi, struct e1000_adapter, napi);
    struct e1000_hw *hw = &adapter->hw;
    struct net_device *poll_dev = adapter->netdev;
    int tx_cleaned = 1, work_done = 0;

    adapter = netdev_priv(poll_dev);

    if (adapter->msix_entries &&
        !(adapter->rx_ring->ims_val & adapter->tx_ring->ims_val))
        goto clean_rx;

    tx_cleaned = e1000_clean_tx_irq(adapter);

clean_rx:
    adapter->clean_rx(adapter, &work_done, budget);

    if (!tx_cleaned)
        work_done = budget;

    /* If budget not fully consumed, exit the polling mode */
    if (work_done < budget) {          /*说明接受队列没有报文了，报文数小于期望的值（默认 64）*/
        if (adapter->itr_setting & 3)
            e1000_set_itr(adapter);    /*调整中断频率*/
        napi_complete(napi);           /*从中断 CPU 队列去掉，通过下一次硬中断再次处理接受报文*/
        if (!test_bit(__E1000_DOWN, &adapter->state)) {
            if (adapter->msix_entries)
                ew32(IMS, adapter->rx_ring->ims_val);
            else
                e1000_irq_enable(adapter);
        }
    }
    return work_done;
```

}

4 软中断处理与协议栈

协议栈也是运行在软中断处理中的，一次 NAPI POLL 取报文，直接调用协议栈处理函数，并不会开启其他类型的 OS 例程去运行协议栈。

4.1 NAPI 交给协议栈前对 skb 的操作

调用函数 skb->protocol = eth_type_trans(skb, netdev);
eth_type_trans 得到的协议是以太网头中的协议，
如 ipv4: ETH_P_IP 0x0800; ipv6: ETH_P_IPV6 0x86DD 。

eth_type_trans(skb, netdev); 还对 SKB->data 指针移位，偏移到 3 层起始地方，当然 VLAN 情况也是同样的操作，在协议栈处理函数中作进一步的操作。代码：skb_pull(skb, ETH_HLEN);

4.2 协议栈接手处理函数

```
int netif_receive_skb(struct sk_buff *skb)
{
	struct packet_type *ptype, *pt_prev;
	struct net_device *orig_dev;
	struct net_device *master;
	struct net_device *null_or_orig;
	struct net_device *null_or_bond;
	int ret = NET_RX_DROP;
	__be16 type;

	if (!skb->tstamp.tv64)
		net_timestamp(skb);

	if (vlan_tx_tag_present(skb) && vlan_hwaccel_do_receive(skb))
		return NET_RX_SUCCESS;

	/* if we've gotten here through NAPI, check netpoll */
	if (netpoll_receive_skb(skb))
		return NET_RX_DROP;
```

```
        if (!skb->skb_iif)
                skb->skb_iif = skb->dev->ifindex;

        null_or_orig = NULL;
        orig_dev = skb->dev;
        master = ACCESS_ONCE(orig_dev->master);
        if (master) {
                if (skb_bond_should_drop(skb, master))
                        null_or_orig = orig_dev; /* deliver only exact match */
                else
                        skb->dev = master;
        }

        __get_cpu_var(netdev_rx_stat).total++;
        /*skb 网络层，传输层指针都移到 3 层报文起始地方*/
        skb_reset_network_header(skb);
        skb_reset_transport_header(skb);
        skb->mac_len = skb->network_header - skb->mac_header;

        pt_prev = NULL;

        rcu_read_lock();

#ifdef CONFIG_NET_CLS_ACT
        if (skb->tc_verd & TC_NCLS) {
                skb->tc_verd = CLR_TC_NCLS(skb->tc_verd);
                goto ncls;
        }
#endif
        /*  此处为 抓包使用的接口  AF_PACKET 类型的处理地方*/
        list_for_each_entry_rcu(ptype, &ptype_all, list) {
                if (ptype->dev == null_or_orig || ptype->dev == skb->dev ||
                        ptype->dev == orig_dev) {
                        if (pt_prev)
                                ret = deliver_skb(skb, pt_prev, orig_dev);
                        pt_prev = ptype;
                }
        }

#ifdef CONFIG_NET_CLS_ACT
        skb = handle_ing(skb, &pt_prev, &ret, orig_dev);
        if (!skb)
                goto out;
ncls:
```

```
#endif
	/*桥模式处理*/
	skb = handle_bridge(skb, &pt_prev, &ret, orig_dev);
	if (!skb)
		goto out;
	/*VLAN 处理*/
	skb = handle_macvlan(skb, &pt_prev, &ret, orig_dev);
	if (!skb)
		goto out;

	/*
	 * Make sure frames received on VLAN interfaces stacked on
	 * bonding interfaces still make their way to any base bonding
	 * device that may have registered for a specific ptype.   The
	 * handler may have to adjust skb->dev and orig_dev.
	 */
	null_or_bond = NULL;
	if ((skb->dev->priv_flags & IFF_802_1Q_VLAN) &&
		(vlan_dev_real_dev(skb->dev)->priv_flags & IFF_BONDING)) {
		null_or_bond = vlan_dev_real_dev(skb->dev);
	}

	type = skb->protocol;
	/*此处根据协议类型，查找上抛到对应 3 层协议栈处理*/
	list_for_each_entry_rcu(ptype,
			&ptype_base[ntohs(type) & PTYPE_HASH_MASK], list) {
		if (ptype->type == type && (ptype->dev == null_or_orig ||
		     ptype->dev == skb->dev || ptype->dev == orig_dev ||
		     ptype->dev == null_or_bond)) {
			if (pt_prev)
				ret = deliver_skb(skb, pt_prev, orig_dev);
			pt_prev = ptype;
		}
	}

	if (pt_prev) {
		/*3 层协议栈处理*/
		ret = pt_prev->func(skb, skb->dev, pt_prev, orig_dev);
	} else {
		kfree_skb(skb);
		/* Jamal, now you will not able to escape explaining
		 * me how you were going to use this. :-)
		 */
		ret = NET_RX_DROP;
```

```
	}

out:
	rcu_read_unlock();
	return ret;
}
```

5 三层协议栈注册

5.1 协议栈注册

```
void dev_add_pack(struct packet_type *pt)
{
	int hash;

	spin_lock_bh(&ptype_lock);
	if (pt->type == htons(ETH_P_ALL))
		list_add_rcu(&pt->list, &ptype_all);
	else {
		hash = ntohs(pt->type) & PTYPE_HASH_MASK;
		list_add_rcu(&pt->list, &ptype_base[hash]);
	}
	spin_unlock_bh(&ptype_lock);
}

struct packet_type {
	__be16			type;	/* This is really htons(ether_type). */
	struct net_device	*dev;	/* NULL is wildcarded here	     */
	int			(*func) (struct sk_buff *,
					 struct net_device *,
					 struct packet_type *,
					 struct net_device *);
	struct sk_buff		*(*gso_segment)(struct sk_buff *skb,
						int features);
	int			(*gso_send_check)(struct sk_buff *skb);
	struct sk_buff		**(*gro_receive)(struct sk_buff **head,
					       struct sk_buff *skb);
	int			(*gro_complete)(struct sk_buff *skb);
	void			*af_packet_priv;
	struct list_head	list;
};
```

5.2 IPv4 协议栈注册

在 static int __init inet_init(void)中注册 packet_type。

```
static struct packet_type ip_packet_type __read_mostly = {
	.type = cpu_to_be16(ETH_P_IP),
	.func = ip_rcv,
	.gso_send_check = inet_gso_send_check,
	.gso_segment = inet_gso_segment,
	.gro_receive = inet_gro_receive,
	.gro_complete = inet_gro_complete,
};
```

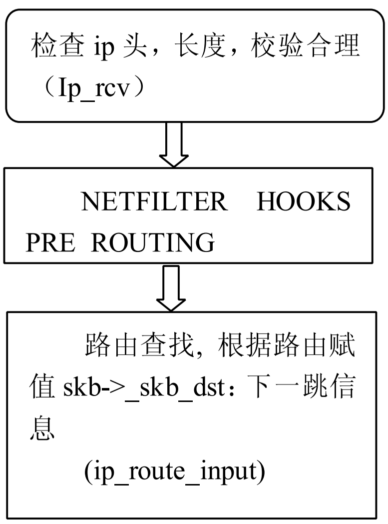
5.3 IPv6 协议栈注册

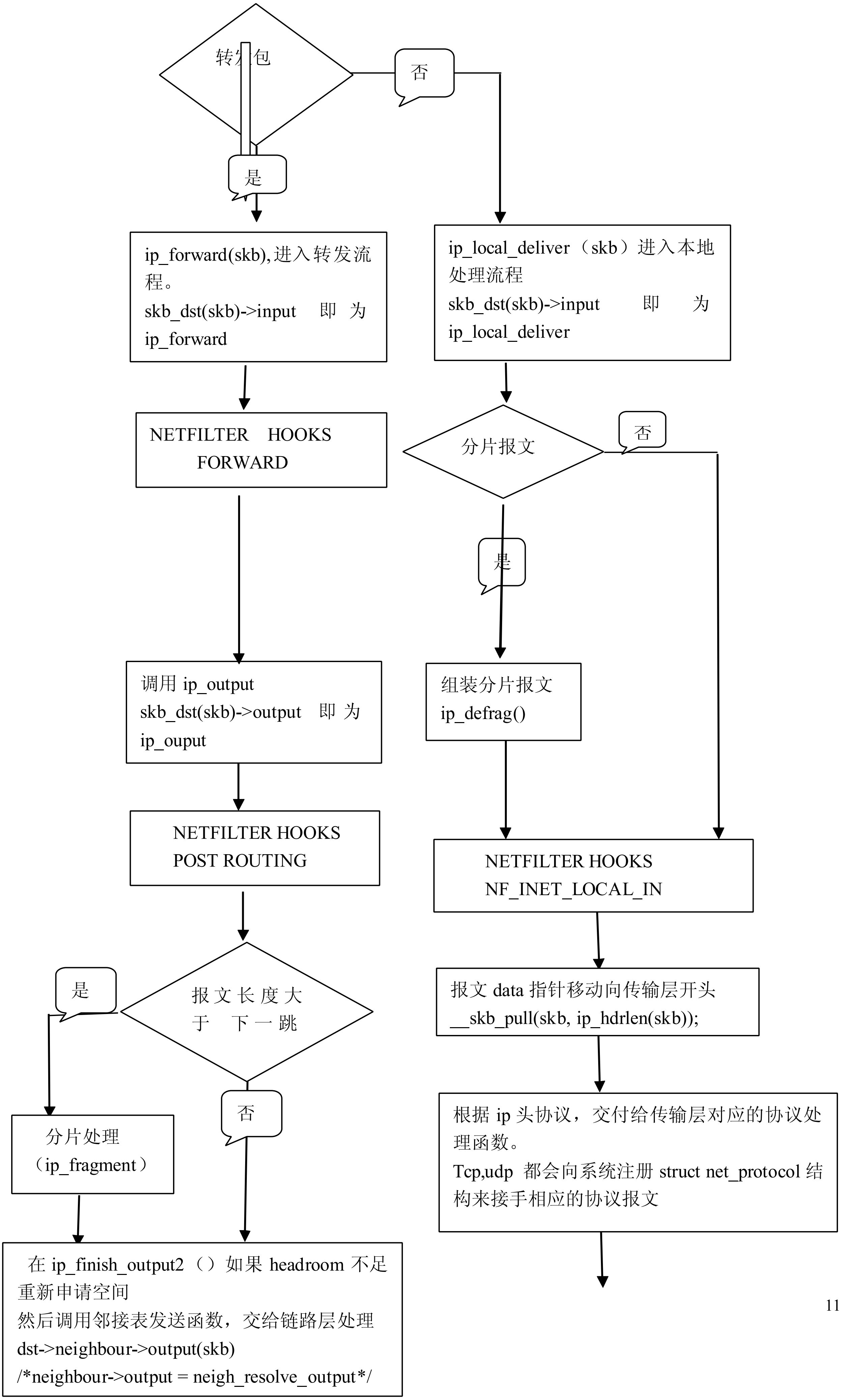
在函数 static int __init inet6_init(void)中注册

```
static struct packet_type ipv6_packet_type __read_mostly = {
	.type = cpu_to_be16(ETH_P_IPV6),
	.func = ipv6_rcv,
	.gso_send_check = ipv6_gso_send_check,
	.gso_segment = ipv6_gso_segment,
	.gro_receive = ipv6_gro_receive,
	.gro_complete = ipv6_gro_complete,
};
```

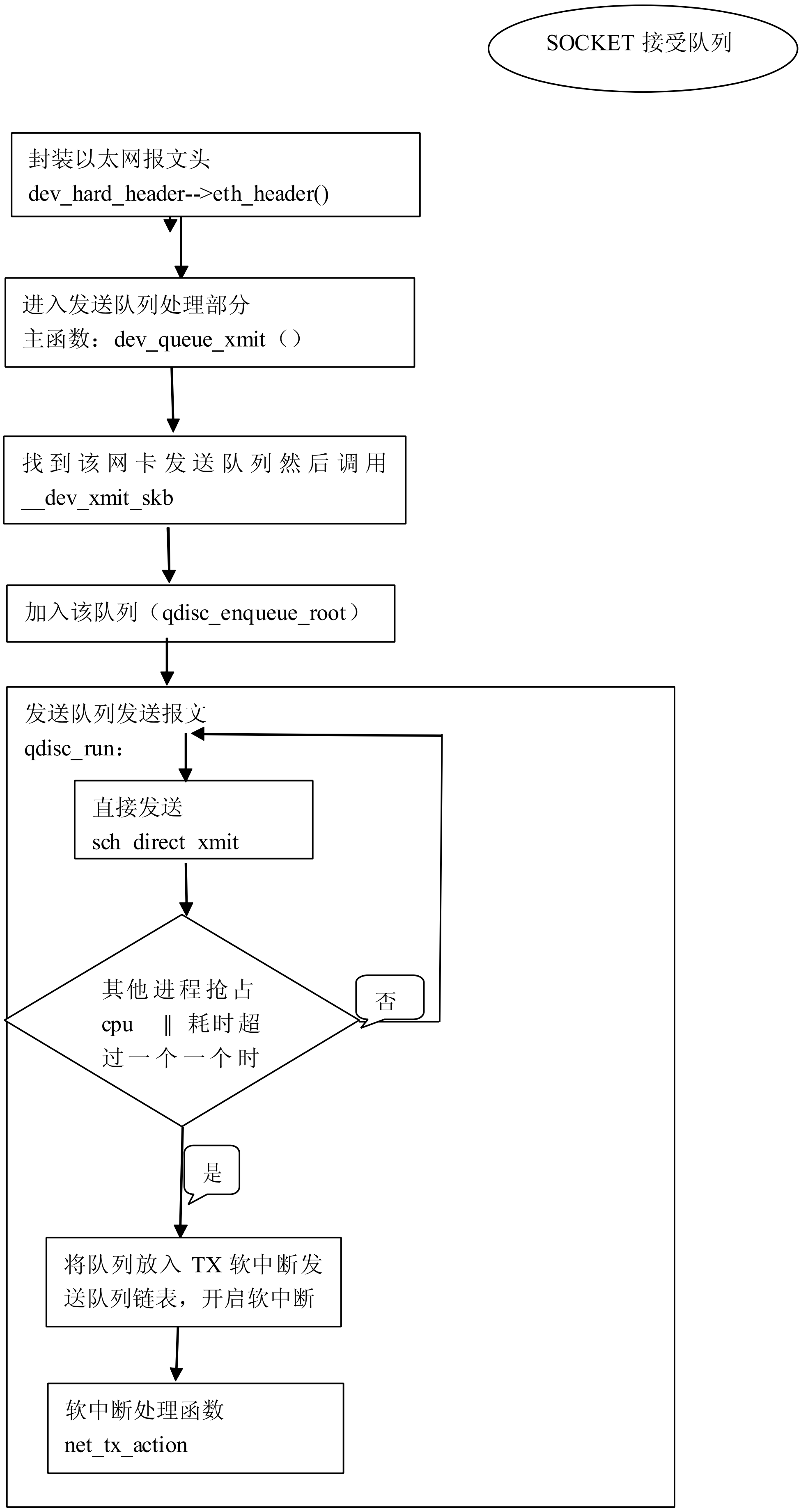
6 IPv4 协议栈处理

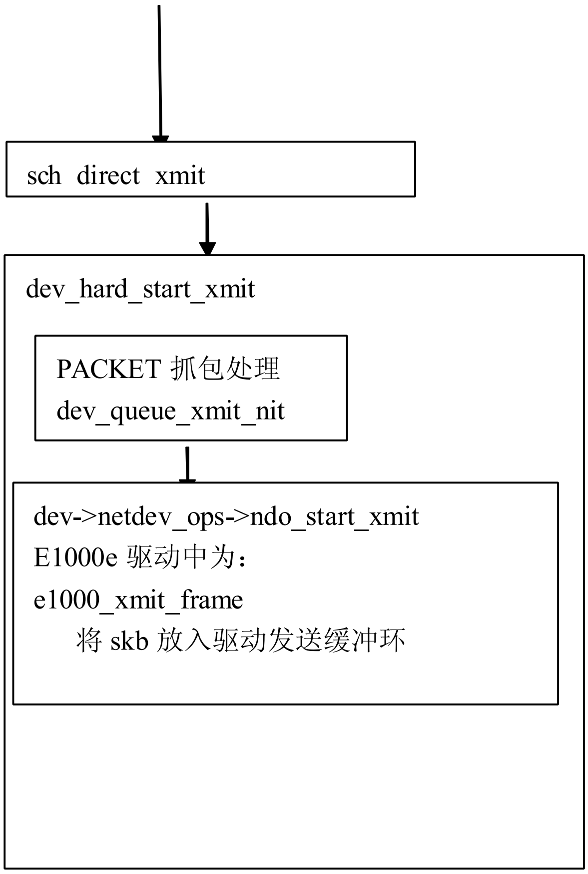
6.1 IPv4 一级流程图





转发包
否
是
ip_forward(skb),进入转发流程。
skb_dst(skb)->input 即 为 ip_forward
ip_local_deliver（skb）进入本地处理流程
skb_dst(skb)->input 即 为 ip_local_deliver
NETFILTER HOOKS FORWARD
分片报文
否
是
调用 ip_output
skb_dst(skb)->output 即 为 ip_ouput
组装分片报文
ip_defrag()
NETFILTER HOOKS
POST ROUTING
NETFILTER HOOKS
NF_INET_LOCAL_IN
报 文 长 度 大 于 下 一 跳
是
否
报文 data 指针移动向传输层开头
__skb_pull(skb, ip_hdrlen(skb));
分片处理
（ip_fragment）
根据 ip 头协议，交付给传输层对应的协议处理函数。
Tcp,udp 都会向系统注册 struct net_protocol 结构来接手相应的协议报文
在 ip_finish_output2（）如果 headroom 不足重新申请空间
然后调用邻接表发送函数，交给链路层处理
dst->neighbour->output(skb)
/*neighbour->output = neigh_resolve_output*/


SOCKET 接受队列
封装以太网报文头
dev_hard_header-->eth_header()
进入发送队列处理部分
主函数：dev_queue_xmit（）
找到该网卡发送队列然后调用
__dev_xmit_skb
加入该队列（qdisc_enqueue_root）
发送队列发送报文
qdisc_run：
直接发送
sch direct xmit
其他进程抢占
cpu ‖ 耗时超
过一个一个时
否
是
将队列放入 TX 软中断发
送队列链表，开启软中断
软中断处理函数
net_tx_action



6.2 IPv4 路由查找 ip_route_input

6.2.1 数据结构介绍

路由 cache 结构：

```
struct rtable {
	union {
/*下一跳的信息，包含 neighbour */
		struct dst_entry	dst;
	} u;

	/* 路由缓存的关键字结构，相当于查询某个东西的 id*/
	struct flowi		fl;
	/*入口设备,该结构主要表示接口 3 层的东西，如 IP*/
	struct in_device	*idev;

	int			rt_genid;
	unsigned		rt_flags;
```

```
/*路由类型， RTN_BROADCAST, RTN_LOCAL,RTN_UNREACHABLE....*/
	__u16			rt_type;
 /*路由目的地址 */
	__be32			rt_dst;	/* Path destination	*/
/*路由源地址*/
__be32			rt_src;	/* Path source		*/
/*入口接口 index*/
	int			rt_iif;

	/* Info on neighbour，网关地址 */
	__be32			rt_gateway;

	/* Miscellaneous cached information */
	__be32			rt_spec_dst; /* RFC1122 specific destination */
	struct inet_peer	*peer; /* long-living peer info */
};

struct dst_entry {
	struct rcu_head		rcu_head;
	struct dst_entry	*child;
	struct net_device	*dev;
	short			error;
	short			obsolete;
	int			flags;
#define DST_HOST		1
#define DST_NOXFRM		2
#define DST_NOPOLICY		4
#define DST_NOHASH		8
	unsigned long		expires;

	unsigned short		header_len;	/* more space at head required */
	unsigned short		trailer_len;	/* space to reserve at tail */

	unsigned int		rate_tokens;
	unsigned long		rate_last;	/* rate limiting for ICMP */

	struct dst_entry	*path;
	/*neighbour 信息*/
	struct neighbour	*neighbour;
/* neigh_hh_output, 用以加快发送速度，缓存 eth 头 */
	struct hh_cache		*hh;
#ifdef CONFIG_XFRM
	struct xfrm_state	*xfrm;
#else
```

```
	void		*__pad1;
#endif
/*路由后，调用的函数，转发为 forward 期间调用的函数*/
	int		(*input)(struct sk_buff*);

/*外发报文调用的函数，转发时对应 postrouing 处理部分*/
int		(*output)(struct sk_buff*);

	struct	dst_ops		*ops;

	u32		metrics[RTAX_MAX];

#ifdef CONFIG_NET_CLS_ROUTE
	__u32		tclassid;
#else
	__u32		__pad2;
#endif



	/*
	 * Align __refcnt to a 64 bytes alignment
	 * (L1_CACHE_SIZE would be too much)
	 */
#ifdef CONFIG_64BIT
	long		__pad_to_align_refcnt[1];
#endif
	/*
	 * __refcnt wants to be on a different cache line from
	 * input/output/ops or performance tanks badly
	 */
	atomic_t	__refcnt;	/* client references	*/
	int		__use;
	unsigned long	lastuse;
	union {
		struct dst_entry *next;
		struct rtable	*rt_next;
		struct rt6_info	*rt6_next;
		struct dn_route	*dn_next;
	};
};
```
区别一个路由的缓冲的关键结构体

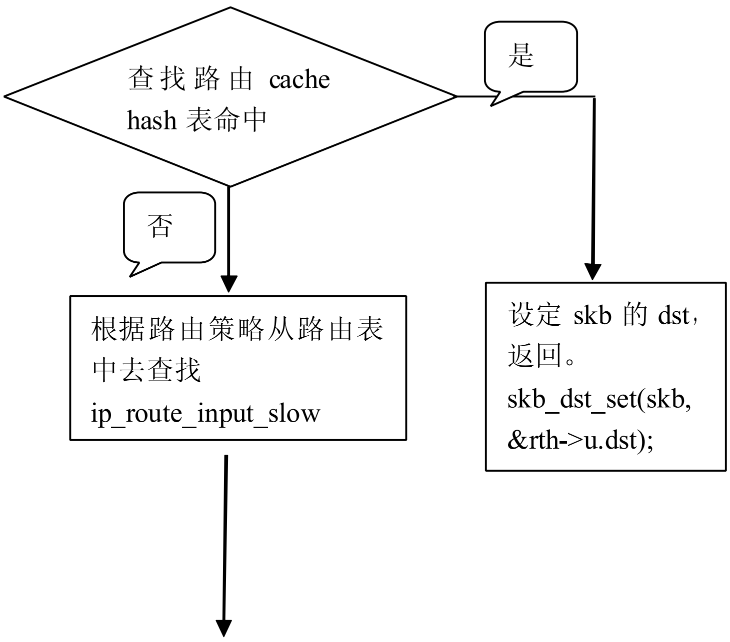
```
struct flowi {
	/*出口，通常为 0*/
	int	oif;
```

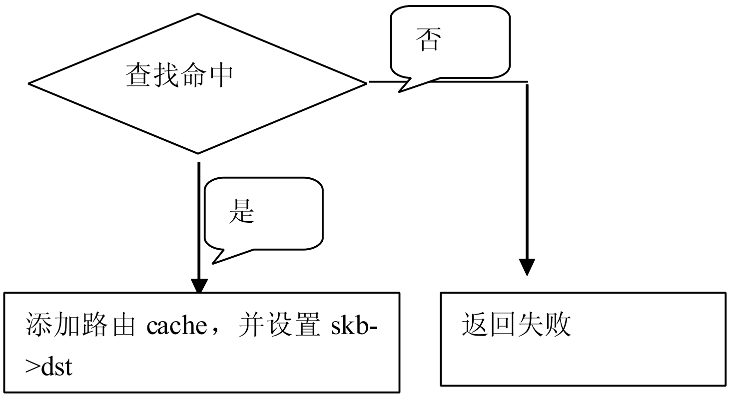
```
    /*入口接口*/
    int    iif;
    /*skb 中的 mark*/
    __u32       mark;

    union {
        struct {
            /*源目的地址，tos 值，scope 值*/
            __be32              daddr;
            __be32              saddr;
            __u8                tos;
            __u8                scope;
        } ip4_u;

        struct {
            struct in6_addr             daddr;
            struct in6_addr             saddr;
            __be32              flowlabel;
        } ip6_u;

        struct {
            __le16              daddr;
            __le16              saddr;
            __u8                scope;
        } dn_u;
    } nl_u;
#define fld_dst             nl_u.dn_u.daddr
#define fld_src             nl_u.dn_u.saddr
#define fld_scope    nl_u.dn_u.scope
#define fl6_dst             nl_u.ip6_u.daddr
#define fl6_src             nl_u.ip6_u.saddr
#define fl6_flowlabel       nl_u.ip6_u.flowlabel
#define fl4_dst             nl_u.ip4_u.daddr
#define fl4_src             nl_u.ip4_u.saddr
#define fl4_tos             nl_u.ip4_u.tos
#define fl4_scope    nl_u.ip4_u.scope

    __u8        proto;
    __u8        flags;
#define FLOWI_FLAG_ANYSRC 0x01
    union {
        struct {
            __be16    sport;
            __be16    dport;
```

```
		} ports;

		struct {
			__u8	type;
			__u8	code;
		} icmpt;

		struct {
			__le16	sport;
			__le16	dport;
		} dnports;

		__be32		spi;

		struct {
			__u8	type;
		} mht;
	} uli_u;
#define fl_ip_sport	uli_u.ports.sport
#define fl_ip_dport	uli_u.ports.dport
#define fl_icmp_type	uli_u.icmpt.type
#define fl_icmp_code	uli_u.icmpt.code
#define fl_ipsec_spi	uli_u.spi
#define fl_mh_type	uli_u.mht.type
	__u32			secid;	/* used by xfrm; see secid.txt */
} __attribute__((__aligned__(BITS_PER_LONG/8)));
```

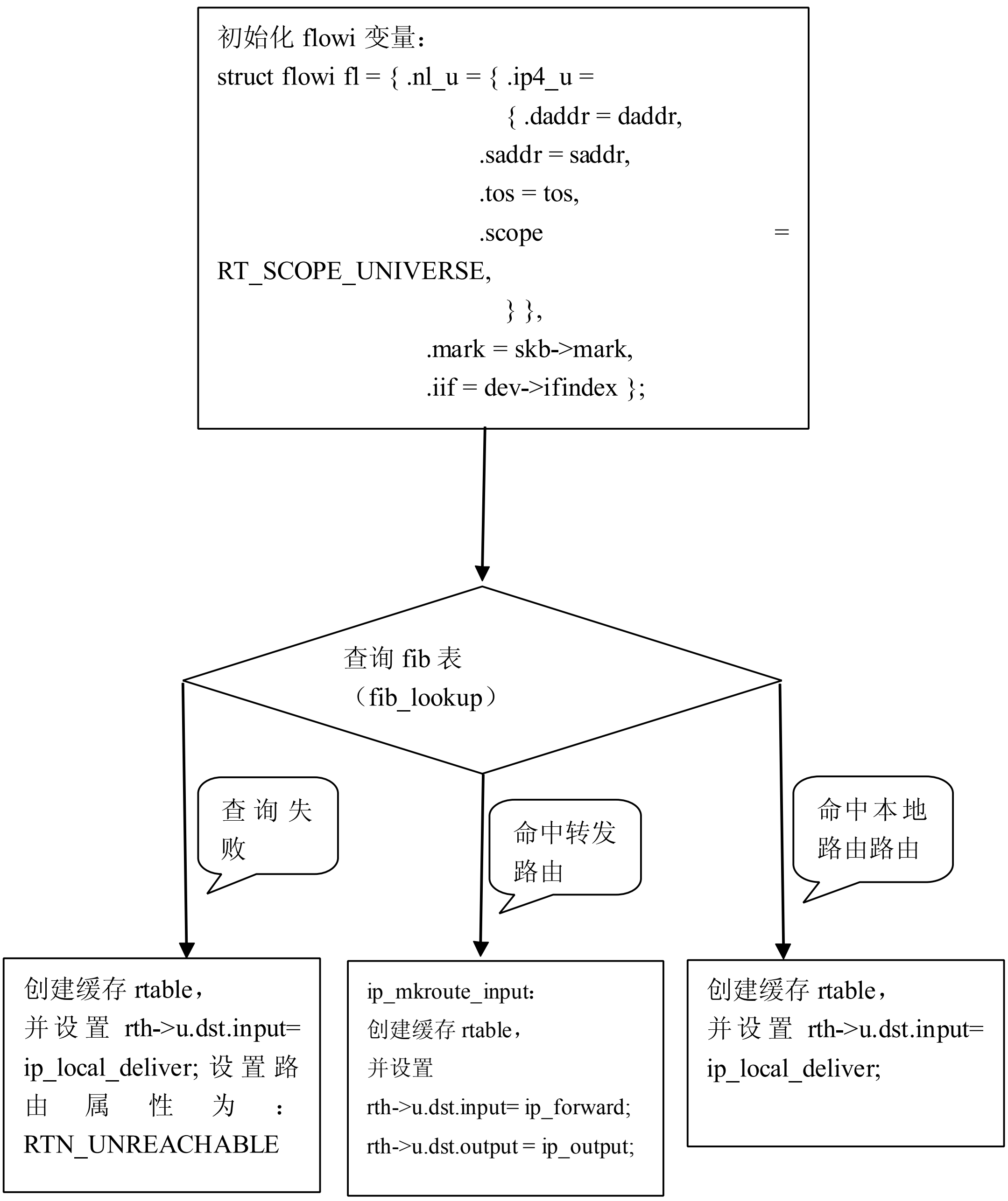
6.2.2 流程介绍

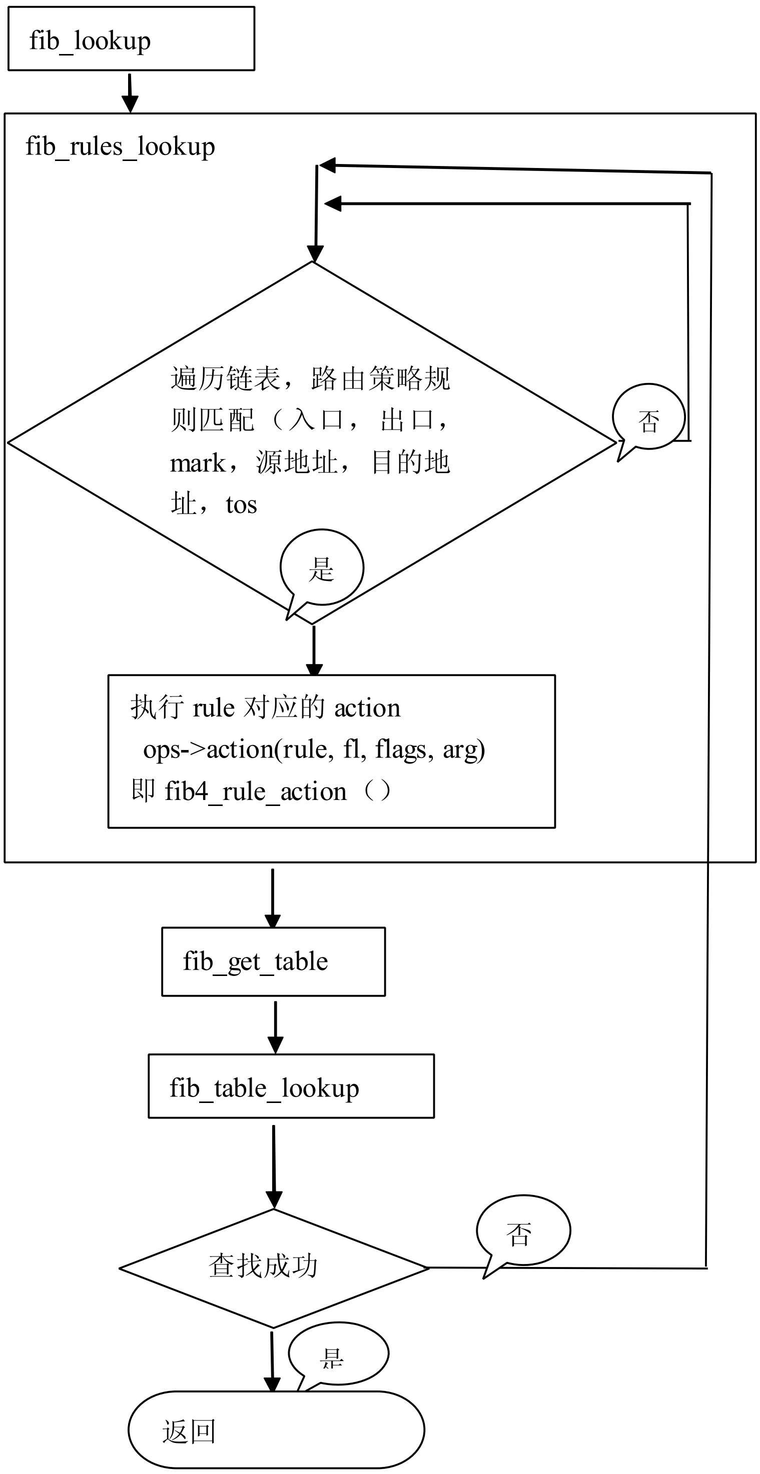
本地接收的报文路由主函数：ip_route_input





路由策略与路由表查询函数
ip_route_input_slow:




fib_lookup
fib_rules_lookup
遍历链表，路由策略规则匹配（入口，出口，mark，源地址，目的地址，tos
否
是
执行 rule 对应的 action
ops->action(rule, fl, flags, arg)
即 fib4_rule_action（）
fib_get_table
fib_table_lookup
查找成功
否
是
返回

fib_table_lookup：从某个路由表查询路由
路由表内的路由条目的组织介绍：
按照网段来分区管理

```
struct fib_table {
	struct hlist_node tb_hlist;
	u32		tb_id;	/*表的 id*/
	int		tb_default;
	unsigned char	tb_data[0]; /**/
};
struct fn_hash *t = (struct fn_hash *)tb->tb_data;
```

路由表分 33 个区 ipv4 地址 32bit，0-32 个区。fn_hash 同时提供了数组合链表两种形式的。这样按区段长度来存储，有利于最长匹配算法的实现。

```
struct fn_hash {
	struct fn_zone *fn_zones[33];
	struct fn_zone *fn_zone_list;
};

struct fn_zone {
	struct fn_zone		*fz_next; /* Next not empty zone	*/
	struct hlist_head	*fz_hash;	/*  该区 hash 表指针	*/
	int			fz_nent;	/* Number of entries	*/

	int			fz_divisor;	/* Hash divisor		*/
	u32			fz_hashmask;	/* (fz_divisor - 1)	*/
#define FZ_HASHMASK(fz)		((fz)->fz_hashmask)

	int			fz_order;	/* Zone order		*/
	__be32			fz_mask;
#define FZ_MASK(fz)		((fz)->fz_mask)
};
```

fib_node 是每个 ip 区段路由的数据结构，挂载在 fn_zone 的 hash 表链表内。

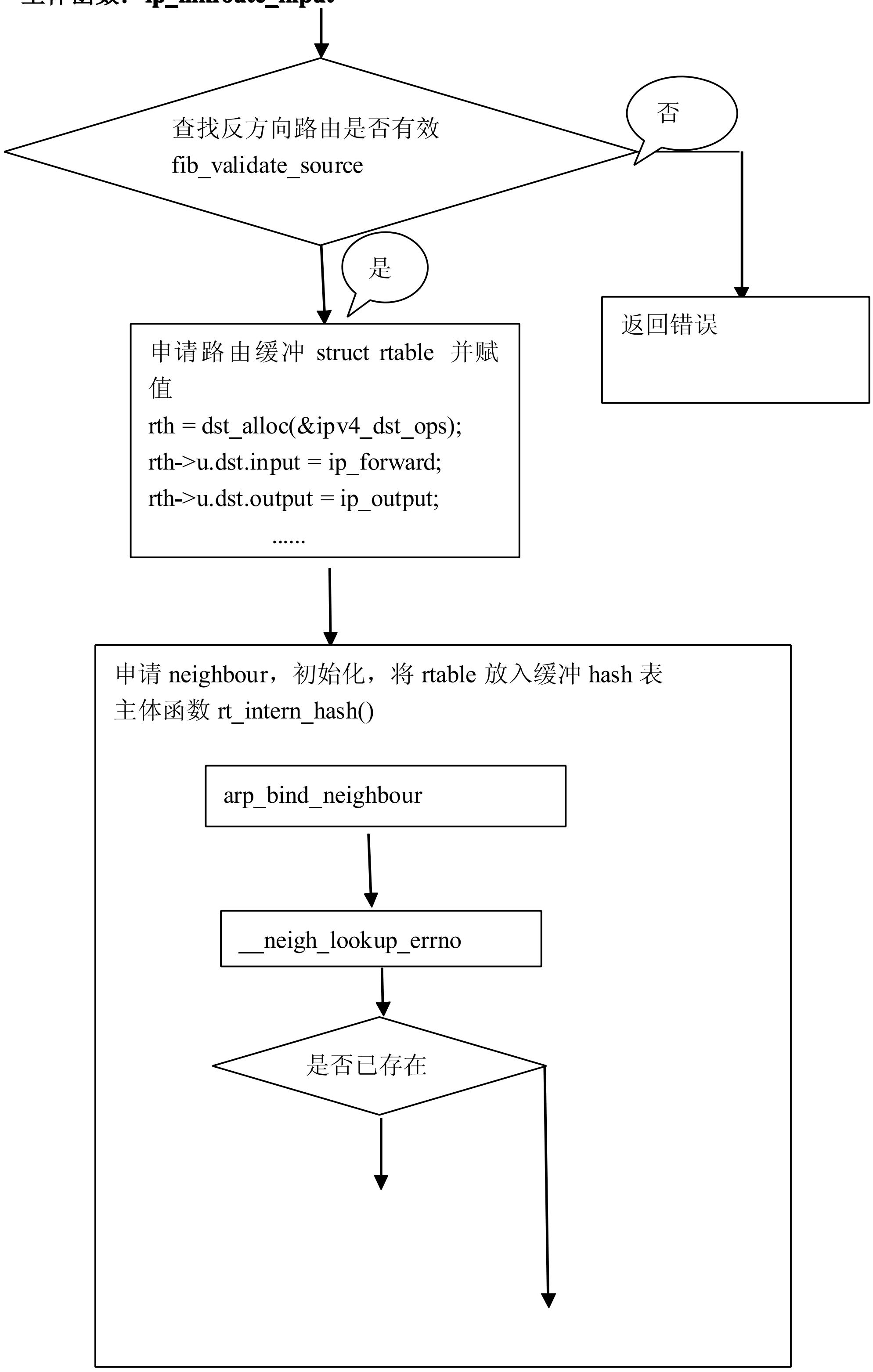
```
struct fib_node {
	struct hlist_node	fn_hash;
	struct list_head	fn_alias;/*同一目的区段的路由项目以链表为组织形式*/
	__be32			fn_key;	/*举例：如果路由为 ip route add 172.16.20.0/24 dev eth0 via 172.16.20.1	，此值为 172.16.20.0 对应的无符号整数*/
	struct fib_alias		fn_embedded_alias;
};
```

```
struct fib_alias {
	struct list_head	fa_list;
	struct fib_info		*fa_info;
	u8			fa_tos;	/*同目的，不同 tos 意味着不同的路由*/
	u8			fa_type;
	u8			fa_scope;
	u8			fa_state;
#ifdef CONFIG_IP_FIB_TRIE
	struct rcu_head		rcu;
#endif
};

struct fib_info {
	struct hlist_node	fib_hash;
	struct hlist_node	fib_lhash;
	struct net		*fib_net;
	int			fib_treeref;
	atomic_t		fib_clntref;
	int			fib_dead;
	unsigned		fib_flags;
	int			fib_protocol;
	__be32			fib_prefsrc;
	u32			fib_priority;
	u32			fib_metrics[RTAX_MAX];
#define fib_mtu fib_metrics[RTAX_MTU-1]
#define fib_window fib_metrics[RTAX_WINDOW-1]
#define fib_rtt fib_metrics[RTAX_RTT-1]
#define fib_advmss fib_metrics[RTAX_ADVMSS-1]
	int			fib_nhs;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
	int			fib_power;
#endif
	struct fib_nh		fib_nh[0];	/*下一跳信息*/
#define fib_dev		fib_nh[0].nh_dev
};

/*路由的下一跳*/
struct fib_nh {
	struct net_device	*nh_dev;
	struct hlist_node	nh_hash;
	struct fib_info		*nh_parent;
	unsigned		nh_flags;
	unsigned char		nh_scope;
#ifdef CONFIG_IP_ROUTE_MULTIPATH
```

```
	int			nh_weight;
	int			nh_power;
#endif
#ifdef CONFIG_NET_CLS_ROUTE
	__u32			nh_tclassid;
#endif
	int			nh_oif;
	__be32			nh_gw;
};
```

下面是构造路由缓冲和 neighbour 节点的构造

主体函数：ip_mkroute_input



创建 neighbour
neigh_create

将 rtable 放入 hash 链表中
rt->u.dst.rt_next = rt_hash_table[hash].chain;
rcu_assign_pointer(rt_hash_table[hash].chain,

7 neighbour 相关介绍

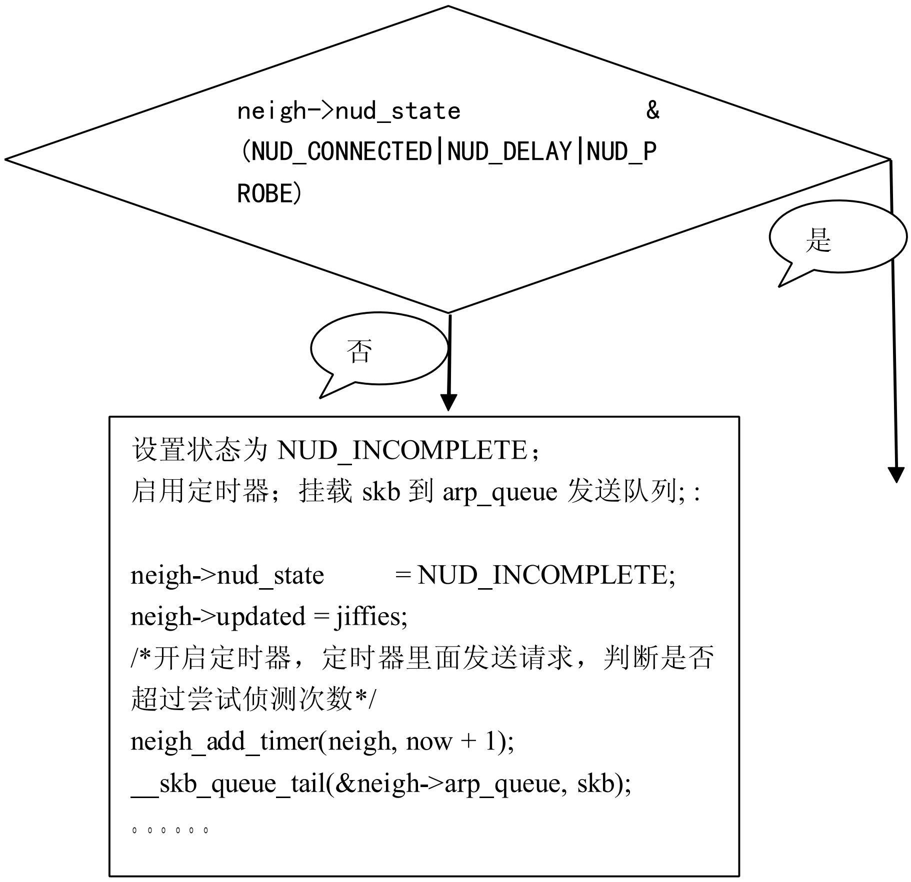
7.1 路由部分调用函数 neigh_create

申请内存空间部分初始化
neigh_alloc：
skb_queue_head_init(&n->arp_queue);
rwlock_init(&n->lock);
n->updated = n->used = now;
n->nud_state = NUD_NONE;
n->output = neigh_blackhole;
n->parms = neigh_parms_clone(&tbl->parms);rwlock_init(&n->lock);
n->updated = n->used = now;

协议相关的初始化：tbl->constructor(n)
Ipv4 为 arp_constructor: e1000e 驱动 header_ops 为 eth_header_ops
if (dev->header_ops->cache)
    neigh->ops = &arp_hh_ops; /e1000e 赋值为该/
else
    neigh->ops = &arp_generic_ops
neigh->output = neigh->ops->output; /*即为：neigh_resolve_output*/
/*在 4.2 介绍过，3 层协议处理完了就交给 neigh->ops->output 来发送*/

放入 hash 链表
n->next = tbl->hash_buckets[hash_val];
tbl->hash_buckets[hash_val] = n;

7.2 neighbour 发送函数 neigh_resolve_output



7.3 定时器处理函数 neigh_timer_handler

```
if (neigh->nud_state & (NUD_INCOMPLETE | NUD_PROBE)) {
        struct sk_buff *skb = skb_peek(&neigh->arp_queue);
        /* keep skb alive even if arp_queue overflows */
        if (skb)
            skb = skb_copy(skb, GFP_ATOMIC);
        write_unlock(&neigh->lock);
        neigh->ops->solicit(neigh, skb);
        atomic_inc(&neigh->probes);
        kfree_skb(skb);
    }
```

Ipv4 对应个 solicit 函数：void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)

该函数将根据 skb 来发送 arp 请求

```
__be32 target = *(__be32*)neigh->primary_key;
arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
             dst_ha, dev->dev_addr, NULL);
```

7.4 arp 报文接收处理函数 arp_rcv

```
static struct packet_type arp_packet_type __read_mostly = {
     .type =    cpu_to_be16(ETH_P_ARP),
     .func =    arp_rcv,
};
```

不再细述